Union – Find. Aplicații. Algoritmul lui Kruskal

Definiţie

Fie mulţimea A={1, 2, ... , n}, n>1. Sistemul S={S_1, S_2,..., S_k}, k>1, constituie o *partiţie* a mulţimii A dacă sunt îndeplinite următoarele condiţii :

1. $S_i \subseteq A$, $\forall$ i$\in${1, 2, ... , k}
2. $S_i \neq \emptyset$, , $\forall$ i$\in${1, 2, ... , k}
3. $S_i \cap S_j = \emptyset$, $\forall i \neq j$, i, j $\in${1, 2, ..., k}
4. $S_1 \cup S_2 \cup \ldots \cup S_k = A$.

Dată fiind mulţimea A partiţiile S_1, S_2, ... , S_k, problema constă în a proiecta o structură de date care să permită executarea eficientă a următoarelor două operaţii fundamentale:

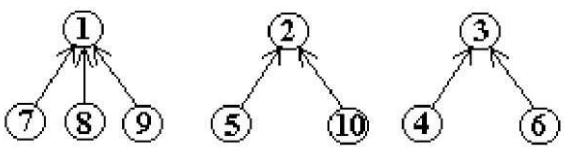
- find(x): determină mulţimea S_i căreia îi aparţine elementul x, x$\in$A;
- union (S_i, S_j): reuneşte mulţimea S_i cu mulţimea S_j, obţinându-se un nou element al mulţimii S, ce va înlocui S_i şi S_j.

De exemplu, dat fiind un graf neorientat, componentele conexe ale grafului constituie o partiţie a mulţimii vârfurilor grafului. Un algoritm de determinare a componentelor conexe ale unui graf neorientat poate fi formulat cu ajutorul operaţiilor *union-find* astfel:

Pentru$\forall$ i$\in${1, 2, ... , n}, S_i = {i};
Pentru $\forall$ [i, j] muchie în graf
    x = find(i);
    y = find(j);
    dacă x!=y atunci union(x, y);

O soluţie eficientă este reprezentarea mulţimilor disjuncte cu ajutorul arborilor cu rădăcină. Fiecare arbore reprezintă o mulţime, iar fiecare nod din arbore un element al mulţimii. Rădăcina arborelui va fi elementul reprezentativ al mulţimii.

De exemplu, să considerăm n=10 şi o partiţie a mulţimii {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} formată din S_1={ 1, 7, 8, 9}, S_2 = {2, 5, 10}, S_3={3, 4, 6}. Partiţia S={S_1, S_2, S_3} poate fi reprezentată ca o pădure formată din trei arbori:

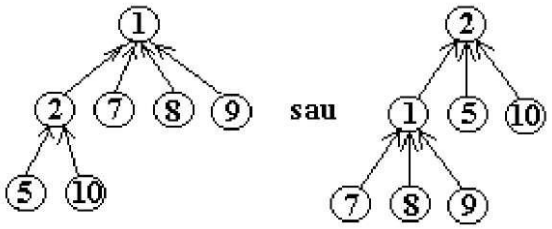


Reprezentăm arborii prin referinţe ascendente, deci pentru fiecare nod din arbore, cu excepţia rădăcinii, reţinem legătura spre părintele său: t(x) = nodul părinte al lui x, sau 0, dacă x este rădăcina arborelui.

Operaţia *find(x)* constă în a determina rădăcina arborelui al cărui nod este x.

```
int find(int x)
{
    while (t[x]!=0)
        x=t[x];
    return x;
}
```

Operaţia *union* se poate realiza transformând unul din arbori în subarborele celuilalt. Reuniunea $S_1 \cup S_2$ poate fi reprezentată în două moduri:



Deci rădăcina unuia din arbori devine părintele rădăcinii celuilalt arbore.

```
void union(int x, int y)
{
    t[x]= y;
}
```

Aceşti algoritmi sunt foarte simpli, dar nu sunt eficienţi. Să considerăm, de exemplu partiţia S={ {1}, {2}, ..., {n}}. Deci configuraţia iniţială constă dintr-o pădure în care fiecare arbore este format doar din rădăcină: t(i)=0, $\forall$ i$\in${1, 2, ..., n}. Să considerăm acum următoarea secvenţă de operaţii *union şi find*: *union(1,2), find(1), union(2,3), find(1), union(3,4), find(1), ..., find(1), union(n-1,n).*

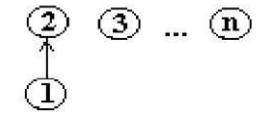
Această secvenţă conduce la următorul arbore degenerat din figura alăturată.

Cum timpul necesar operaţiei *union* este constant, cele n-1 operaţii *union* au timpul de execuţie de O(n). Dar fiecare operaţie *find (1)* parcurge tot drumul de la nodul 1 până la rădăcină. Cum timpul de execuţie necesar operaţiei *find* pentru un nod de pe nivelul i este O(i), obţinem un timp de execuţie de $O(n^2)$ pentru cele n-2 operaţii *find(1).*

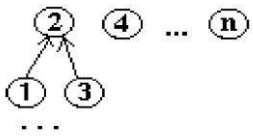
Putem îmbunătăţi eficienţa operaţiilor *union* şi *find*, aplicând următoarea *regula de ponderare* pentru *union(x, y)*: dacă numărul de niveluri din arborele cu rădăcina x este mai mic decât numărul de niveluri din arborele cu rădăcina y, atunci y va deveni părintele lui x, altfel x va deveni părintele lui y.

Utilizând regula de ponderare secvenţa de operaţii *union* şi *find* de mai sus va conduce la următorii arbori:

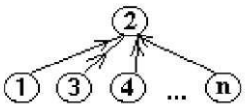
union(1,2):



union(2,3):



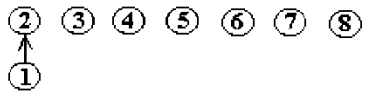
union(n-1,n)

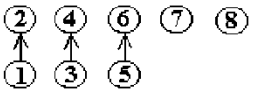


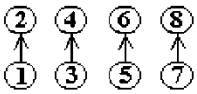
În acest caz timpul de execuţie necesar operaţiilor *find* este O(n), deoarece arborii obţinuţi au înălţimea cel mult egală cu 1. Totuşi, există cazuri mai defavorabile.

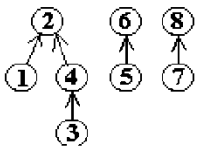
De exemplu, fie n=8. Iniţial vom avea pădurea:

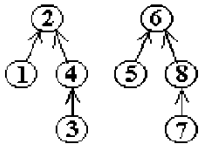
union(1,2)

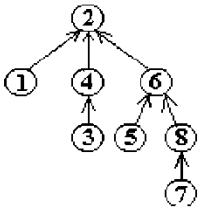












Înălţimea arborelui obţinut este $3=\log_2 8$.

Dacă A este un arbore cu n noduri obţinut în urma aplicării operaţiei *union* folosind regula de ponderare, înălţimea arborelui A este cel mult egală cu [log n].

Demonstraţie

Vom proceda prin inducţie după n, numărul de noduri. Pentru n=1, rezultatul este evident.

Presupunem afirmaţia adevărată pentru arbori cu i noduri (1<i<n-1), obţinuţi în urma aplicării algoritmului *union*. Să demonstrăm că înălţimea oricărui arbore cu n noduri A_n, obţinut în urma aplicării algoritmului *union* este cel mult egală cu [log n].

Fie *union (j, k)* ultima operaţie *union* executată pentru obţinerea arborelui A_n. Notăm cu m numărul de noduri din arborele cu rădăcina j. Arborele cu rădăcina k are m-n noduri. Putem presupune, fără a restrânge generalitatea, că 1<m<n/2. Deci înălţimea h a arborelui A_n va fi egală cu înălţimea arborelui cu rădăcina k sau cu o unitate mai mare decât înălţimea arborelui cu rădăcina j .

În primul caz: h<[log (n-m)]<[log n].

În cel de-al doilea caz: h<[log m]<[log (n/2)]+1<[log n].

Pentru a aplica regula de ponderare este necesar ca pentru fiecare arbore să cunoaştem numărul de niveluri. Fie r, un vector în care r[i] reprezintă numărul de niveluri din arborele cu rădăcina i. Funcția *union*, în urma aplicării regulii de ponderare devine:

```
void union(int x, int y)
{
        if(r[x]<r[y])
                t[x]=y;
        else
                if(r[x]>r[y])
                        t[y]=x;
                else
                {
                        t[y]=x;
                        r[x]++;
                }
}
```

Rangul unei rădăcini va fi mai mare decât rangul unui nod din interior, iar valoarea rangului nu va reține cu exactitate adâncimea arborelui, dar arborii mai mari vor avea valoarea r mai mare și atunci regula de ponderare se aplică corect.

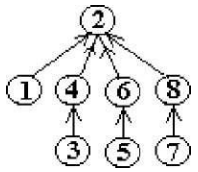
Se garantează un timp de execuţie de O(log n) pentru operaţia *find(x),* pentru orice nod x dintr-un arbore obţinut prin operaţii *union* folosind regula de ponderare.

Pentru a îmbunătăţi în continuare timpul de execuţie a operaţiei *find(x),* vom utiliza următoarea *regula de compresie:* orice nod y de pe drumul de la rădăcina arborelui la nodul x va deveni fiu al rădăcinii arborelui. Funcția *find* devine:

```
int find(int x)
 {
        int rad=x;
        while(t[rad]!=0)
                rad=t[rad];
        int tmp=x;
        while(t[x]!=0)
        {
                tmp=t[x];
                t[x]=rad;
                x=tmp;
        }
}
```

Funcţia *find* parcurge drumul de la nodul x la rădăcină de două ori, o dată pentru determinarea rădăcinii, a doua oară pentru comprimarea drumului de la x la rădăcină. Totuşi, această modificare reprezintă o îmbunătăţire, deoarece într-o secvenţă de operaţii *union - find,* timpul total de execuţie va fi mai mic.

Să considerăm arborele obţinut prin secvenţa de operaţii *union* din exemplul precedent Executând prima oară operaţia *find (8)* obţinem arborele alăturat:



Au fost necesare trei deplasări pe legături ascendente pentru a identifica rădăcina arborelui şi apoi, pentru compresia drumului de la rădăcină la nodul 8 alte două deplasări. Dar următoarele apeluri ale funcţiei *f ind (8)* vor necesita o singură deplasare până la rădăcina arborelui. Astfel costul total al unei secvenţe de operaţii *f ind (8)* va fi mai mic.

Aplicații

1. Problema *păduri de mulțimi disjuncte* de pe infoarena.ro

Problema este o aplicație imediată a algoritmului *union-fiind*

```
#include <fstream>
#define dim 100020
using namespace std;

/*initial toate punctele se afla intr un arbore cu un singur element
//fiecare arbore va avea inaltimea 0
//padurile se vor reprezenta prin vectorul de tati t, iar r va retine inaltimea fiecarui arbore
//radacina va avea in vectorul de tati valoarea 0
//n -nr de noduri, m numarul de operatii de tip union find, union este codificata cu 1, find cu 2
*/

int t[dim],r[dim],n,m;

void uniune(int x,int y)
{
   /*aplicam uniunea a doi arbori cu regula de ponderare,
   arborele cu rang mai mic se uneste la arborele cu rang mai mare,
   radacina arborelui mai mic va avea ascendent radacina arborelui mai mare
```

```
    evident rangul arborelui mai mare nu creste prin uniune
    daca rangul este egal inseamna ca rangul noului arbore va fi cu 1 mai mult
    */
    if(r[x]>r[y])
        t[y]=x;
    else
        if(r[x]<r[y])
            t[x]=y;
        else
        {
            t[y]=x;
            r[x]++;
        }
}

int find(int x)
{
    int rad=x;
    //caut radacina arborelui in care se afla x
    while(t[rad]!=0)
        rad=t[rad];
    //cand se iese din while rad va fi radacina arborelui in care se afla x
    //in continuare aplic compresia drumului, adica orice nod non radacina dintr un arbore va fi legat direct la radacina
    int tmp;
    while(t[x]!=0)
    {
        tmp=t[x];
        t[x]=rad;
        x=tmp;
    }
    return rad;
}

int main()
{
    int mod,x,y,r1,r2;
    ifstream fin("disjoint.in");
    ofstream fout("disjoint.out");
    fin>>n>>m;
    while(m>0)
    {
        fin>>mod>>x>>y;
        if(mod==1)
        {
            //unim arborii in care se afla x si y, doar daca nu au aceeasi radacina
            r1=find(x);
            r2=find(y);
            if(r1!=r2)
                uniune(r1,r2);
        }
        else
        {
            if(find(x)==find(y))
                fout<<"DA\n";
            else
                fout<<"NU\n";
        }
        m--;
    }
    fin.close();
    fout.close();
    return 0;
     }
```

2. Obținerea componentelor conexe ale unui graf neorientat.

Problema *dfs* de pe infoarena. Problema se poate rezolva și cu parcurgerea în adâncime. Regula utilizării *union – fiind* în acest caz: dacă graful este static, se folosește parcurgerea în adâncime, dacă la graf se adaugă/elimină dinamic muchii atunci folosim *union fiind*.

```
#include <fstream>
#include<vector>
#define dim 100020
#define dim2 200020
using namespace std;

vector <vector<int> > comp(dim2);//comp este o matrice cu dim2 linii, fiecare linie este necompletata
int n,m,nr;
int t[dim],r[dim];

int find(int x)
{
   /*caut radacina subarborelui in care se gaseste x si aplic compresia caii pentru toate nodurile incepand de la x
   din acel arbore, nodurile situate sub x nu vor fi compresate!
   */
   int rad=x;
   while(t[rad]!=0)
      rad=t[rad];
   //compresia caii
   int tmp;
   while(t[x]!=0)
   {
      tmp=t[x];
      t[x]=rad;
      x=tmp;
   }
   return rad;
}

void uniune(int rad1,int rad2)
{
   //aplica uniunea a doi arbori, dupa rang, adica arborele cu rang mai mic se uneste la arborele cu rang mai mare,
   //adica radacina arborelui mai mic va avea ascendent radacina arborelui mai mare
   //evident rangul arborelui mai mare nu creste prin uniune
   //daca rangul este egal inseamna ca rangul noului arbore va fi cu 1 mai mult
   //vom uni practic subarborii care au radacina rad1, respectiv rad2
   if(r[rad1]>r[rad2])
      t[rad2]=rad1;
   else
      if(r[rad1]<r[rad2])
         t[rad1]=rad2;
      else
      {
         t[rad2]=rad1;
         r[rad1]++;
      }
}
int main()
{
   ifstream fin("dfs.in");
   ofstream fout("dfs.out");
   int x,y,rad1,rad2,i,j;
   fin>>n>>m;
   //pp numarul de compo conexe ca fiind n
   nr=n;
```

```
for(i=1;i<=n;i++)
    comp[i].push_back(i);

  while(m>0)
  {
    fin>>x>>y;
    /*apelez find ptr amandoua extremitatile, daca au aceeasi radacina fac parte din aceeasi comp conexa
    daca au radacini diferite adaug in componenta conexa a radacinei
    elimin componenta conexa a nodului care se adauga la alta componenta prin stergerea nodului
    ma folosesc de rang pentru a sti la ce componenta conexa adaug
    */
    rad1=find(x);
    rad2=find(y);
    //daca am egalitate nu adaug nimic, nodurile sunt deja incluse
    if(rad1!=rad2)
    {
      if(r[rad1]>=r[rad2])
      {
        /*toata lista lui y se va adauga componentei lui x
        unim
        golim lista lui y
        scadem nr
        */
        for(i=0;i<comp[y].size();i++)
          comp[rad1].push_back(comp[y][i]);
        comp[y].clear();
        uniune(rad1, rad2);
        nr--;
      }
      else
      {
        /*x se va adauga la y
        unim
        golim lista lui x
        scadem nr
        */
        for(i=0;i<comp[x].size();i++)
          comp[rad2].push_back(comp[x][i]);
        comp[x].clear();
        uniune(rad1, rad2);
        nr--;
      }
    }
    m--;
  }
  //nr ne va indica nr de componente conexe
  //parcurgem comp si daca linia nu este vida, listam nodurile componentei
  fout<<nr<<"\n";
  for(i=1;i<=n;i++)
    if(comp[i].empty()==false)
    {
      for(j=0;j<comp[i].size();j++)
        fout<<comp[i][j]<<" ";
      fout<<"\n";
    }

  return 0;
   }
```

3. Arborele parțial de cost minim. Algoritmul lui Kruskal

Se dă un graf conex neorientat G cu N noduri si M muchii, fiecare muchie având asociat un cost. Se cere să se determine un subgraf care cuprinde toate nodurile și o parte din muchii, astfel încât subgraful determinat să aibă structura de arbore și suma costurilor muchiilor care îl formează să fie minim posibilă. Subgraful cu proprietățile de mai sus se va numi arbore parțial de cost minim pentru graful dat.

Kruskal a propus următoarea rezolvare: sortăm muchiile în ordine ascendentă după cost și selectăm în ordine muchiile care nu formează un ciclu în arborele parțial construit. Ne oprim când am ales n-1 muchii sau când am terminat muchiile. Verificarea existenței unui ciclu se face cu ajutorul metodei *union fiind*. Pentru o muchie aplicăm *fiind* pentru ambele extremități. Dacă rădăcinile arborilor din care fac parte sunt diferite înseamnă că adăugarea muchiei nu va produce un ciclu în arborele parțial de cost minim.

Algoritmul poate fi descris astfel folosind un limbaj de tip pseudocod:

citim și memorăm muchiile cu 3 caracteristici: extremitatea inițială, extremitatea finală, cost
sortăm crescător muchiile după cost
aplicăm fiind pentru cele două extremități ale muchiei
dacă rădăcinile sunt diferite
    aplicăm union
    adăugăm muchia la arborele parțial de cost minim
    actualizăm costul

```
#include <fstream>
#include<vector>
#include<algorithm>

#define dim 200020
using namespace std;

/*muchiile vor fi memorate cu o structura: (x,y) muchia, c-costul
se cere apm, folosim Kruskal
retin muchiile in vectorul de tip muchie m, le sortez crescator dupa cost
aleg muchiile in ordine si daca nu se formeaza un ciclu, muchia face parte din apm si o retin in vectorul apm
pentru a verifica daca nu se formeaza cicluri folosesc algoritmul union find si anume
la inceput toate nodurile formeaza o multime(un arbore), iau o muchie si daca extremitatile fac parte din aceeasi arbore,
se formeaza un ciclu, sar peste, altfel adaug la apm si maresc costul. Folosesc find pentru a cauta radacina arborelui si
union pentru a uni subarborii (folosesc comprimarea caii si uniunea dupa regula de ponderare)
*/
//r[i] retine adancimea arborelui
//padurile vor fi memorate cu vectorul de referinte ascendente, t[radacina]=0

int t[dim],r[dim];
int n,m1;//nr de noduri si nr de muchii

struct muchie
{
   int x,y,c;
};
vector<muchie>m,apm;
```

```
int find(int x)
{
   //caut radacina subarborelui in care se gaseste x si aplic compresia caii pentru toate nodurile incepand de la x
   //din acel arbore, nodurile situate sub x nu vor fi compresate!
   int rad=x;
   while(t[rad]!=0)
      rad=t[rad];
   //compresia caii
   int tmp;
   while(t[x]!=0)
   {
      tmp=t[x];
      t[x]=rad;
      x=tmp;
   }
   return rad;
}

void uniune(int rad1,int rad2)
{
   /*aplica uniunea a doi arbori, dupa rang, arborele cu rang mai mic se uneste la arborele cu rang mai mare
   radacina arborelui mai mic va avea ascendent radacina arborelui mai mare
   evident randul arborelui mai mare nu creste prin uniune
   daca rangul este egal inseamna ca rangul noului arbore va fi cu 1 mai mult
   vom uni practic subarborii care au radacina rad1, respectiv rad2
   */
   if(r[rad1]>r[rad2])
      t[rad2]=rad1;
   else
      if(r[rad1]<r[rad2])
         t[rad1]=rad2;
      else
      {
         t[rad2]=rad1;
         r[rad1]++;
      }
}

//sortarea se face dupa cost si atunci ne trebuie o functie care sa spuna lui sort cum se face sortarea
bool compare(muchie a, muchie b)
{
   return a.c<b.c;
}

int main()
{
   ifstream fin("apm.in");
   ofstream fout("apm.out");
   fin>>n>>m1;
   long long cApm=0;//aici voi memora costul apm
   int i,rad1,rad2;//i pentru a cicla de m ori, ext1 retine extremitatea1, resp extremitatea2 cand fac uniune
   muchie h;//retin muchia citita si apoi o copii in vectorul m
   for(i=0;i<m1;i++)
   {
      fin>>h.x>>h.y>>h.c;
      m.push_back(h);
   }
   //sortez crescator vectorul de muchii cu ajutorul fct compare
   sort(m.begin(),m.end(),compare);
   //apm va avea n-1 muchii, le iau in calcul pe toate din m pana cand am ales n-1 muchii
   int nr=1,j=0;
   while(nr<n && j<m1)
```

```
{
   //aplica find pentru extremitatile muchiei si daca nu intoarce aceeasi valoare, adaug in apm si unesc subarborii
   rad1=find(m[j].x);
   rad2=find(m[j].y);
   if(rad1!=rad2)
   {
      cApm+=m[j].c;
      apm.push_back(m[j]);
      uniune(rad1,rad2);
      nr++;
   }
   j++;
}
//afisez costul total si muchiile din apm
fout<<cApm<<"\n"<<n-1<<"\n";
for(i=0;i<apm.size();i++)
   fout<<apm[i].x<<" "<<apm[i].y<<"\n";
fin.close();
fout.close();
return 0;
 }
```